
Computer Supported Cooperative Work **12:** 297–327, 2003.

Recomposition: Coordinating a Web of Software Dependencies

REBECCA E. GRINTER
Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA (E-mail: beki@parc.com)

Abstract. In this paper, I revisit the concept of recomposition – all the work that development organizations do to make sure that their product fits together and into a broader environment of other technologies. Technologies, such as Configuration Management (CM) systems, can ameliorate some of a software development team's need to engage in recomposition. However, technological solutions do not scale to address other kinds of recomposition needs. This paper focuses on various organizational responses to the need for recomposition. By organizational response, I mean how individuals engage in recomposition so that the organization can assemble software systems from parts. Specifically, I describe how those responses are manifested in the day-to-day communications and responsibilities of individuals throughout the organization. I also highlight how changes in an organization complicate recomposition. The paper concludes with a discussion of three features of software development work that are revealed by recomposition: the affects of environmental disturbances on development work, the types of dependencies that require recomposition, and the images of organizations required to manage the recomposition.

Key words: empirical studies, recomposition, software development

1. Introduction

Given the highly collaborative nature of software development, it is not surprising that it has attracted the attention of researchers interested in computer supported cooperative work (CSCW). Most commercial software development requires tens, hundreds and even thousands of people – typically spread among numerous locations – working together to produce the final product. Moreover, since it requires computers to build software the environment is rich with technologies that support development work. These technologies range from purpose-built systems, such as configuration management tools, to general tools such as group calendars. More recently, open source movements have shown that Internet technologies such as e-mail can facilitate development (Mockus, Fielding and Herbsleb, 2000). For CSCW researchers, software development provides opportunities to study interactions among developers that may be mediated by a range of technologies.

For these reasons, and others, researchers have produced a sizable literature about software development practices.[1] Researchers have studied many aspects of software development work. Studies of meetings have highlighted the cognitive

and social aspects of collaborative design work (see for example, Bucciarelli, 1994; Herbsleb et al., 1995). Other studies have examined computerized and paper-based technologies used to coordinate development (see for example, Palen, 1999; Whittaker and Schwarz, 1999). Finally, researchers interested in CSCW and empirical software engineering have conducted longitudinal studies of project coordination that span the entire development life cycle (see for example, Button and Sharrock, 1996; Curtis, Krasner and Iscoe, 1988; Herbsleb and Grinter, 1999; Potts and Catledge, 1996).

In this paper, I revisit the topic of coordinating development. However, I focus on the work necessary to ensure that a software product can be assembled from its component pieces, which I call *recomposition* (Grinter, 1998). Examination of recomposition highlights the social relationships that arise among individuals responsible for distinct, but related, software components. The paper begins by introducing recomposition and its relationship to decomposition.

2. Decomposition and recomposition

2.1. MODULAR DECOMPOSITION AND DIVISION OF LABOUR

When researchers and practitioners met at the first conference about software development – convened by the North Atlantic Treaty Organization (NATO) in 1968 – one discussion topic was coordinating the production of software systems (NATO, 1969). One technique that would emerge as central to coordinating software development was taking form at the same time as that NATO conference. Also in 1968, Melvin Conway published a paper arguing that the structure of software systems tended to mirror the communications structure of the committee that designed it (Conway, 1968).

This relationship between software and the organization that builds it – known today as Conway's Law – would become a cornerstone of the process of modular decomposition. One of the earliest proponents of modular decomposition was David Parnas. Like Conway, Parnas recognized the relationship between a software system and a division of labor. Unlike Conway, Parnas used this relationship to argue how to design software systems. As Parnas said in his paper that described the criteria for systems decomposition:

> In this context "module" is considered to be a responsibility assignment rather than a sub-program. (Parnas, 1972, p. 1054)

For Parnas, modular decomposition was the process of deciding how to break apart a problem into small pieces that individuals could build. He argued that this process would create efficiencies in development because:

> development time should be shortened because separate groups would work on each module with little need for communication. (Parnas, 1972, p. 1054)

These efficiencies would only arise when each module had certain technical properties. Two concepts that became a part of good modular decomposition were

coupling and information hiding. Coupling focuses on the relationships among modules and argues for reducing those dependencies. Specifically, decomposition should produce a design that makes each module as independent as possible from others.

Parnas' concept of information hiding also emphasizes creating an easily understood separation among modules by using interface specifications. Each module has a specified interface that allows other modules to interact with it. Information hiding is the process of designing these interfaces so that other modules can use the functions provided, but do not know how those functions are implemented. The advantage of this approach is that developers only need to know what results the module generates, and what data it needs to be passed, and the rest happens without their knowledge. Information hiding exemplifies "good" modular design because it separates implementation dependencies between modules, one developer does not have to know how a module works to interact with it, they only need to know the function that that module provides and how to pass data to it.

Today, modular decomposition is widely recognized as being the correct approach to software design. Books teaching software engineering to undergraduates advocate modular decomposition because supports the goal of achieving a design that embodies properties such as low coupling (Ghezzi, Jazayeri and Mandrioli, 1991). Moreover, modular decomposition and information hiding have also become practice by becoming embedded in object oriented programming languages (Sommerville, 1996). With contemporary languages such as C++ and Java being object oriented, many people find themselves using modular decomposition to achieve good software design.

With the adoption of modular decomposition came the division of labor organized around modules. However, the focus on this relationship was lost. Software developers and researchers adopted modular decomposition for its technical advantages. Over time the connection between the decomposition of code and the division of labor was largely lost in software engineering writings. Consequently, development proceeds with surprisingly little attention paid to how the division of labor determined by decomposition creates systematic coordination needs that influence the entire development process.

2.2. DEPENDENCIES AND RECOMPOSITION

Despite the widespread adoption of modular decomposition, empirical studies of software development suggest that developers still spend time coordinating their efforts (Curtis et al., 1988; Herbsleb and Grinter, 1999). One study of collocated developers suggests that they spent 15% of their time communicating with colleagues about shared problems (Perry, Staudenmayer and Votta, 1994). More recent studies suggest that the length of time it takes to coordinate development rises sharply when developers are geographically remote. In fact, a study of geographically distributed development suggests that remote developers take

almost three times as long to resolve a problem as their collocated counterparts (Herbsleb et al., 2000).

Empirical evidence suggests that despite modular decomposition techniques developers still need to coordinate with each other. One reason that developers need to communicate and coordinate is that their code contains technical relationships known as *dependencies*. Dependencies among modules can take a number of forms. For example, if module A passes a variable to module B, then module B depends on A to give it that type of information. Another type of dependency occurs when module A has information that B needs in order to compile. Then A must be compiled before B. One of the hardest kinds of dependency to find is when module A generates a certain behavior during program execution which module B needs to function. This last case is known as a run-time dependency, and is hard to see because there may be scant evidence of this relationship in the written code itself.

Dependencies occur among modules for at least four reasons. First, designers define some dependencies during the decomposition process. Modular decomposition did not argue that the need to communicate would disappear. Rather, it argued that the need would be reduced, and those communications would be focused on various aspects of the software. For example, if developers use information hiding they would still need to coordinate with others about the interface specifications.

Second, dependencies arise when software cannot be completely decomposed. Real-time hardware-embedded software systems prove difficult to decompose into modules because of the real-time and hardware constraints imposed on the solution space. For example, to meet real-time requirements it may be necessary to have one module "look inside" another rather than wait for interface communications. This kind of design trade-off – between good design and performance – can lead to increased dependencies among modules. However, outside this domain, this rarely creates problems. In theory, full modular decomposition can be achieved.

Third, legacy code can complicate the decomposition process and increase the number of dependencies. Decomposing new problems onto a base of existing code can be compared to retrofitting or an extending an existing house as opposed to building a new one. Since many products have more than one life cycle – as enhancements and revisions are added over time – so the amount of "legacy code" increases. However, legacy code also means that the design process becomes constrained by what ever already exists. As software systems evolve their purpose can change, making the original design – while good for the original product – less optimal for the current incarnation. In addition, dependencies may arise when the new software has to interact with the old software in order for the final product to remain compatible with older versions of the system already in use.

Fourth, software requirements change during the development life cycle (Curtis et al., 1988). As these requirements change so developers add, modify, and delete pieces of the code. Each time a piece is changed, any dependencies it has with

other pieces can change. Clean and simple interfaces may be transformed into more complicated ones as dependencies accommodating changes in the desired feature set are added. In other words, although dependencies are defined during design they evolve throughout development. Dependencies arise for at least four reasons: as part of decomposition, when problems are not amenable to modularity, if legacy code compromises the process, and as requirements evolve during development. For all these reasons and others software ends up with dependencies among its components (Parnas and Clements, 1986). From the perspective of understanding collaborative work, these dependencies become more interesting when they span modules that are owned by different developers.

When dependencies span modules owned by two or more developers then those developers have to coordinate and communicate their actions with each other well enough so that everyone has a shared understanding of what exactly is being worked on. For example, when two developers own modules that interact – A passes variables to B and B returns a function – then the developer working on A has to know what type of variables B expects to get from A and B has to know what form to return the answer in. This may get specified in the original design, but as A and B work on their respective modules they need to ensure that if they make any changes to the original agreements that they communicate that to the other. If they fail to coordinate their actions sufficiently then one possible consequence is that their modules A and B will not work together. In simple cases they will cause the program to stop running. In more difficult cases they will actually interact but produce strange and unexpected results that may only appear in another part of the program later on.

This work – of coordinating and communicating enough to maintain a shared understanding of the dependency – is recomposition. Developers who share a dependency have to share enough information among the dependent developers so that they all have the same understanding of the problem that they are solving together, how their modules fit into the overall solution, and how this solution affects their individual modules. I call it recomposition to emphasize its relationship to decomposition. Decomposition encourages developers to focus on their own modules having decomposed them from the overall program. Recomposition is all the work that individuals have to do to understand how their modules fit into the overall software system. Moreover, while decomposition may occur once at the beginning of the current development life cycle, recomposition occurs throughout the remainder of the development effort.

2.3. TECHNOLOGY AND RECOMPOSITION

Configuration Management (CM) systems are a technological solution to the challenge of coordinating dependencies (Grinter, 1998; Tellioglu and Wagner, 1997). They help "small groups" of developers to coordinate their work in a variety of ways.[2] In addition to performing an automated assembly of all the code inside

an instance of the system – the build – some CM systems can help developers determine whether their code depends on others.

However, even with a CM tool, developers usually need to spend time coordinating dependencies with their co-workers. For example, to coordinate a change developers must determine what each person has done to their own code, how their revisions influence other people's code, what combined functionality has been produced, and whether that reflects the desired end-state (Grinter, 1998). All of these discussions focus on a coordination work that arises from the need to align modules that must function together as a system – recomposition.

Despite this need to coordinate changes, smaller companies still have a relatively easy time managing dependencies. Small companies with 30 developers may be able to put their code into a couple instances of a CM system (perhaps taking advantage of an architectural separation in their code) and manage the cross subsystems dependencies by the simple fact that all the developers know each other. Tool Corp, who I will introduce in the next section, was one such company.

Companies with bigger development projects have more difficulties coordinating dependencies among subsystems. Some of the difficulties arise from the fact that these companies have more subsystems to manage, and each subsystem can be fairly large itself. On top of those difficulties, Comms Corp and Computer Corp who I introduce in the next section had another challenge in managing their dependencies. They both had projects that had been evolving for upwards of twenty years. Some of their current dependencies stemmed from the fact that they were retrofitting their software to provide functions that it was not originally intended for. But Comms Corp also had young projects that contained dependencies among subsystems, which required considerable coordination.

Both companies had a variety of solutions in place to coordinate their dependencies. Despite their current solutions, managers in both companies were aware of the challenges of dependencies that spanned subsystems. Moreover, they were interested in improving the overall process. As two different managers put it:

> We're thinking of starting an initiative that would be a whole task force or specially chartered group to examine dependencies. Just because you know that is such a hard problem for people because it bites, its enterprise wide dependencies, right so how do you manage them, well right now we don't manage it, we stumble over it, and try to solve it every dependency one at a time. Senior Manager, Computer Corp.
>
> Our problem is beyond the scope of a single database. We're going to need upwards of ten possibly . . . and we don't have visibility across databases as we speak today. Manager, Computer Corp.

In this paper, I describe the current solutions that Computer and Comms Corp used to coordinate dependencies. Specifically, I describe how individuals took on informal and more formal communications and roles to manage recomposition and how successful these strategies were.

The rest of the paper is organized as follows. First, I describe the sites and methods used. Then I describe the two ways of managing recomposition: communications and organizational roles and groups. Finally, I show how any way of managing recomposition can be complicated by changes within the organization itself. The paper concludes with a discussion about what recomposition comprises and how individuals in the organization conceive of the organization in order to engage in recomposition.

3. Sites and methods

3.1. SITES

The data presented in this paper come from three companies. I used two criteria for site selection. First, the companies had to produce software commercially. I wanted to study corporations who sold their software – either "shrink-wrapped" or bundled with hardware – and had customers for their products. Second, these corporations had to be willing to host a researcher, sometimes for extended periods of time.

Tool Corp is the vendor of a configuration management (CM) tool. At the time, they were finishing a new release of the product as well as an upgrade to an existing version. Both products contained code from prior systems. During my time there Tool Corp. grew from 14 to 18 developers and their product consisted of 1 million lines of code (LOC). I spent three months on site and conducted over 100 interviews there.

Computer Corp is a large computer company that produces a real-time operating environment. They employ 700 developers who are distributed across a number of sites in the US and other continents. Their product suite consists of around 10 million LOC. I conducted 14 interviews with developers and spent three days on site.

Comms Corp is a telecommunications equipment company that builds and sells a range of telephony products. In this paper I draw on data gathered on three products Alpha, Beta, and Shape. The products span a range of telecommunications functions. Beta was a large system with approximately 50 million LOC, and the other two were smaller systems that contained approximately 2 million LOC. I spent approximately six months gathering data on each project. Data was largely drawn as an active participant on these projects, where I consulted on integration procedures. My role involved attending meetings with a range of individuals across the project, conducting some interviews especially where there were gaps in my knowledge and participating in integration planning activities.

3.2. METHODS

A qualitative approach was used to gather and analyze data from all the sites. Data were collected using forms of observation and interviewing. At Tool Corp

and Comms Corp. I conducted participant observation, non-participant observation, and interviews. The participant observation did not consist of writing code, instead I helped by conducting usability analysis, reviewing system architectures, facilitating project retrospectives, and process design. At Computer Corp my site access was limited to interviewing and the only observation I conducted was while I remained on site.

Most of the interviews were semi-structured. They relied on an interview guide that directed questioning towards recomposition, but allowed the individuals time to talk about their own coordination problems. In addition to observation and interviewing, I also gathered supplemental materials. These materials included project documentation, organizational information, and systems overviews.

I used the Grounded Theory method of developing explanations of practice from qualitative data (Strauss and Corbin, 1990). In this case, I wanted to know more about how large software projects are coordinated, who by, and using what technologies and procedures. Grounded Theory consists of cycles of data gathering followed by analysis. The analysis drives the next cycle of data gathering by extending, modifying, and contradicting the current explanation of the phenomena under investigation. Data gathering ceases when new cycles fail to change the explanation, which then becomes the Grounded Theory.

Moreover, as Strauss argued, it is possible to then take the newly developed Grounded Theory of events, and try and apply it to understanding other data gathered using Grounded Theory methods (Strauss, 1987). I visited the first two sites described – plus several others not discussed here – to learn about the phenomena that would eventually become the concept of recomposition. I entered Comms Corp, the third site described, with the goal of learning whether recomposition work permeated other parts of the development process. I also gathered data at Comms Corp that extended and revised my Grounded Theory of recomposition itself.

4. Using communication to coordinate dependencies in larger projects

One approach to managing the difficulties of coordinating software dependencies at assembly time was to communicate about them prior to assembly. Communications about dependencies was a method to try and prevent the assembly process from breaking by discussing changes. In this section I describe two approaches used to communicate changes: broadcast and meetings.

4.1. BROADCAST

Broadcast was the crudest method either company used to coordinate dependencies. Specifically, developers would broadcast changes they made to their code to others who depended on their work. The broadcaster hoped that other developers

would make any necessary adjustments to their code. Broadcast typically involved e-mailing out notifications about changes that could affect others.

> We use e-mail and some more informal things where you just know because you've been around a long time. You tell everyone that you know who would be impacted that they're getting impacted. Developer, Computer Corp.

Broadcast was a quick, informal way to communicate changes among individuals who knew each other. However, its usefulness was contingent on knowing everyone to contact. As one developer described:

> I'm creating code and I don't know who uses it, I know a few people cause they complain when its broken, but there are always more like if I send out a mail message saying well we're obsoleteing this kernel component we're not going to produce anymore then people come out of the woodwork you know I get totally shocked at the number of developers that over time you know just sort of this has infiltrated into other subsystems. Developer, Computer Corp.

While developers on small projects could send out "global" e-mail to the entire project, this was not acceptable for larger projects. For the largest projects, global e-mail would have meant sending notes to upwards of 5000 people. In addition to consuming corporate network bandwidth, many developers would have regarded the message as spam e-mail, and that deterred developers from sending these messages out.

4.2. MEETINGS

Projects at Computer Corp and Comms Corp did not use broadcast very often. A much more common approach to communicating about dependencies prior to software assembly was to have a meeting. Meetings have had a long history of being used as a genre for reporting information about the state of work (Yates, 1989). Both Comms Corp and Computer Corp used some meetings to report about the state of the development work and feed that information up. It was those same reporting meetings that were used as occasions to discuss problems in the state of work created by dependencies.

The Network Project at Comms Corp illustrates the use of meetings to coordinate dependencies. The Network Project consisted of 3000 developers who worked on a collection of elements that collectively provided a type of telecommunications network. The architecture of these elements was distributed, each providing a specialized piece of the overall function. The 3000 developers were organized into departments. Each department would have responsibility for one part of one element of the network, since the parts were big enough that they would require 300 developers and a department consisted of roughly 150 people.

Alpha was one part of one element in the Network Project and a corresponding department of developers. I joined the department head of Alpha for her meetings. The first one was a meeting of the department head and her team leaders (each

team leader had approximately 7–15 development staff that reported to him, and the department head had 10 team leaders that reported to her). Each team leader owned several software functions that their team implemented. Collectively, the team leaders owned code that interoperates with each other, to produce functions that the department head had responsibility for delivering to the Network Project. Field notes from two of these meetings illustrate how dependencies got coordinated:

> 8:30 a.m. Alpha Weekly Meeting. Everyone has a two-page spreadsheet that contains a list of items in 8pt font. Each row is a system function; each column a version of the system the department head has responsibility for. The purpose of the meeting is to review the status of each function with respect to each version of the system. A few of the boxes are already filled in, letting everyone know whether the function is implemented for the release. The goal of this meeting is to fill in the rest of the boxes.
>
> 10:30 a.m. The meeting is not finished, but must be adjourned, since the department head has another meeting. The spreadsheet remains incomplete. The discussions have been of several forms. Some team leaders announce that they have delivered functions. The box gets marked "ready." Software that does not work receives more discussion. In some cases, the software depends on hardware or other software from other groups outside the department. The department head takes an action item to "rattle some cages" to ensure that her schedule does not get delayed by non-functional code any further. The cases where one team leader depends on another team leader's code and there are problems (non-delivery, technical incompatibility between the two groups) get two responses. One response is a lengthy discussion of the technical difficulties of the current work. This often gets the other team leaders into longer debates about different technical solutions that might be tried. Others are not discussed, the team leader acknowledges that he has spoken with either just the team leader, or with the leader and the department head. Nothing is written in the box. Nothing more is said publicly.
>
> 10:40 a.m. Network Project Monthly Meeting. The department head arrives at the monthly project status review meeting. The meeting consists of all the department heads and their boss, the Vice President. The purpose of this meeting is to coordinate the progress of Network Project. The department head distills the information from the previous meeting to an appropriate level for this meeting. She lines up her items for discussion. Unlike the previous meeting there is no spreadsheet for discussion and not all the people are physically present in the room, instead there are about 20 people in the room and an unspecified number on conference call – including the Vice President – from other states and other countries.
>
> 12:30 p.m. The meeting ends. The Vice President wraps up. Each department head has delivered a summary of the progress of the software. The department head for Alpha describes the technical decisions that her department

is considering. She mentions delays that come from outside her department when they refer to hardware or software shipments that come from outside the corporation as a whole. This garners widespread support from her colleagues, many of who have also been waiting for a certain hardware delivery. Delays from sources outside her department, but within the span of the Vice President's management, are not mentioned in the course of the meeting. Instead, she approaches these department heads and has brief conversations with them, either in the hallway afterwards or quietly in the back of the room during the meeting itself. She explains to me that the corporate culture does not approve of formal problem escalation until peer resolution fails to work. I do not see any cases where informal problem resolution has failed, but she tells me some "war stories" about when that does happen.

The purpose of both meetings was to discuss progress made towards implementing various software components. In both meetings, software progress was discussed in terms of delays to progress. The nature of the delays was sometimes independently hard problems that someone, a team leader or a department head just needed to work though. Many times delays resulted from some form of dependency. Sometimes it was delays in receiving other components that held up progress. Sometimes the delays arose because a developer was making progress on their own work and then for reasons they didn't fully understand they were receiving messages from people testing their code that it did not function anymore. Often the process of debugging previously working code lead to the realization that someone else had changed their code.

In the case of working software, Alpha's weekly meeting provided a forum for team leaders to announce successful completion of software for a release. For the other team leaders it was an opportunity to find out whether they could use that software in their own testing to ensure that their code worked together. In other words, it was a means for the teams to align their individual efforts with other parts of the Alpha product.

In the case of non-working software, the weekly meeting was also an opportunity for each team leader to engage the department head in resolving dependencies that were outside the department. When those dependencies came from outside the corporation it was an opportunity for the department head to make a note of them for discussion at the monthly meeting. When those dependencies came from another department working on the Network Project it was an occasion for the department head to work on a solution. In this particular instance the department head had an immediate opportunity to do so because she attended the monthly immediately afterwards. When the monthly review did not immediately follow the weekly review, she would return to her office and begin scheduling individual appointments with her peers to discuss delays. She did this face-to-face over lunch when she could, but in some cases her peer was in another state or continent and a phone call had to do.

The weekly meeting also forced other kinds of dependency resolution or recomposition work among the team leaders. Team leaders (and the department head) knew that the "corporate culture" of the company meant that they should speak with their counterpart before engaging the department head. The process of engaging a more senior manager was known as "escalation" and implied that all possible negotiations among peers had not resolved the dependency, and that there were problems with that dependency that had to be made visible to management.

Sometimes team leaders self-reported that they were delaying other people with particularly difficult problems. Open admissions tended to occur when a delay had only just emerged. Early admission, with a precise technical description of the complexity of the problem, often appeared to encourage the other team leaders who depended on this code to help out with suggestions for possible design solutions.

The Network Project's monthly meeting shared two common features with the Alpha weekly meeting. First, the department heads were comfortable discussing delays that came from outside the corporation. Second, the same "corporate culture" also kept department heads from discussing delays in their own progress bought about by delays or difficulty with another department head's code. Again, it would have constituted "escalation."

However, the two meetings also differed. In the Network Project monthly meeting no-one seemed to openly discuss difficulties that were caused by delays that stemmed from another part of the Network Project. These were discussed privately. The Network Project meeting also had a lack of cohesion, which arose from the combination of having some people present and others on conference call. This, of course, facilitated the necessary private in-meeting discussions among department heads that shared a need to coordinate a dependency discretely.

Both meetings were a significant part of the process of coordinating change across subsystems owned by different team leaders and different department heads. For team leaders, the weekly project meeting caused them to talk with other team leaders to try and resolve problems. Those team leaders would then communicate back to individual developers that problems were communicated (in the case of the person reporting the problem) and that a problem needed resolution (in the case of the developer who's code was related to the person reporting the problem).

The monthly meetings were an opportunity for Alpha's department head to discuss dependencies with the other department head that owned the other side of the dependency in question. Based on discussions with her peer she would get information that she could then take back to her department and report down to the relevant team leader. However, the Alpha department head did not use the weekly project meeting to discuss the results of any discussions with other department heads. She always discussed the conclusions with individual team leaders. This was because sometimes it was Alpha department code that was the source of a delay, and by discussing the problem privately she was avoiding making that team leader's code "visible."

Other projects in Comms Corp used meetings as a vehicle to communicate and coordinate dependencies that spanned subsystems. Meetings, either public and routine or private and individually scheduled, were used to ensure that the message was communicated appropriately. By appropriately I mean that using meetings for recomposition followed a set of expectations about how changes that span subsystems owned by different individuals are managed. Rather than making these changes highly visible, they need to be handled carefully, ensuring that good vertical and lateral communications can be established, so that the problem can be effectively resolved. "War stories" – which many people on the Network Project and in Comms Corp – could tell me were a mechanism for ensuring that people, such as myself, knew why these protocols existed.

5. Organizational roles to coordinate change

Both Computer Corp and Comms Corp also coordinated software development activities by instituting various positions and groups within the organization. Some of these groups had the "official" role of coordinating development work and others did not. In this section, I examine the official and less official roles that the organizations developed as a mechanism for coordinating dependencies among software.

5.1. BUILD AND RELEASE GROUPS

People who performed build and release functions had roles assigned to them by the organization that included coordinating dependencies. A "build manager" was responsible for pulling together all the software for a subsystem. They were typically assigned to projects where the code base was large enough that a configuration management tool could not hold it all. The process of pulling all the code together required that the build manager gather all the latest code from every developer on the project, compile it together, and test it to see whether it worked.

> When I worked in different groups that were larger we had a build manager who did that for us and that was ideal in terms of when we said we were done and we released our code and he captured it they put it some place and then secured it. Developer, Computer Corp.

Beta was a product that had some extremely large components that were beyond the scope of tool support. Beta, a project at Comms Corp, was a growing project. It started with a few hundred people, but at the time of the study had at least 3000 headcount assigned. One part of Beta, Betalite, was sufficiently big that they also had a build manager to coordinate their build.

The work to coordinate the build itself begins with considerable coordination between the build manager and the development staff as the following excerpt from field notes illustrates:

9:30 a.m. The build manager for Betalite begins this week's build process. Rather than starting with compilation it starts with e-mail sent to the entire staff of Betalite. Then he picks up a pencil and notepad and leaves his office. We begin the process of walking around the development offices. At each office that is occupied we stop and talk with the developer. The conversations take the form of the developer informing the build manager what new code they know about, including code of their colleagues who are away from their desks. Most importantly the developers inform the build manager about whose changes he must incorporate into the build if their own changes are going to work. In other words they flag dependencies between their code and their coworkers. The build manager takes notes. These notes consist of locations of files, new functionality that has been produced since the previous build, and the names of people who own any related code.

We spend the rest of the day walking around the building talking to developers and crosschecking these dependencies as well as gathering new ones. The day ends when the build manager returns to his office. He checks his e-mail to see whether anyone else has sent notes regarding their work. Scanning through his notes he determines what will go into the build. He takes all the latest changes, except for those where it has been flagged that there is a dependency but he has not been able to contact the other person or people who share that dependency. He makes notes to inform them that their changes have not made the build. His last acts are to set the build running and leave for the evening.

The following day

7:00 a.m. The build has failed. The build manager looks through the error logs generated to see whether he can determine what piece of code is responsible. He then looks though the code that had been running at the time the build broke to see whether he can spot the error. Nothing seems visible. Next, he contacts the developer whose code was running when the build broke. He tells me that he always starts with the person who owns the code, because even if it turns out it's not their fault, the developer knows that part of the system well and may have good leads on where the problem has arisen.

9:00 a.m. We find the developer whose code was running when the build broke. He joins us in the build manager's office. They start to discuss the broken build. The developer clearly does not want the problem to be in his own code – there's a stigma associated with being the person who broke the build – but at the same time he also wants to know what the problem is since he may have to make some changes to his own code. The build manager and the developer spend some time discussing the logs and looking at the code. Eventually they decide that the problem is that half a change from a related piece of code has gotten into the build.

> 10:00 a.m. A third developer arrives. This third developer joins the diagnosis discussion about the problem. The three eventually conclude that the broken build is a result of a mis-communication and no one is to blame. They had not synchronized when the change would enter the build.

Or as a manager at Computer Corp succinctly put it:

> What do they do? They just going around trying to figure out what the configuration is, and how to build it, and why this build failed, that's all they do. Manager, Computer Corp.

The build function involved two kinds of coordination. First, the build manager had to coordinate with each developer what should and should not be in the build. Second, the build manager had to coordinate the resolution of any problems with the build. Like Alpha developers and managers in the previous section, the build managers at Comms Corp, and Computer Corp, did not initially raise the problem beyond the developers concerned. Creating visibility for a broken build has social implications in many companies.[3]

Further, by resolving the problem locally, the build manager got more information about the cause of the difficulty. Behind a small problem was the opportunity to gather more knowledge about the current structure of the product. The build manager could use this knowledge in future build cycles.

Tool Corp, while small enough to manage their builds largely through automated processes, also had a build manager. At Tool Corp the build manager was responsible for determining why a build broke. Although the automated systems did a comprehensive job of reporting the point at which the system had failed – not just what module was running, but what the "state" of the program was and what the values of variables were if it failed during the testing after compilation – their build person still needed to follow up with the developer responsible.

Tool Corp's management used the build manager function as a training tool and always assigned a new person to the role. It was a training opportunity because the new person would, in the course of tracking down the reasons for broken builds, meet many of the developers working on the project. The new person would also come to learn the overall structure of the code and begin to understand the dependencies that existed.

Tool Corp's build manager had two functions, overseeing the build and the release. This happened because each time that the build manager compiled and tested the code they were in fact pulling together everything required for a general release of the system. At Computer and Comms Corp build managers did not gather together enough subsystems to comprise a release, and both companies had other, distinct, organizational groups specifically focused on release management.

Computer Corp produced one large product and had one release group. It was a department of between 15–30 people who gathered together all the subsystems that would become the final product for shipping to the customer.

> Because right now development does all of its things and then it throws over a very high fence to release and then release has a whole other different process to managing these configurations, and literally they have only one version at any given point in time and then the next release window happens they get an entirely different version and if the first one is completely overlaid. Manager, Computer Corp.

At Comms Corp each large product had an individual release group. Some of the larger products had multiple release groups each focused on bringing together more parts of the overall product. Beta, for example, had three levels of release management.

As Beta grew in size and importance within the company a number of the release functions were formalized. The process of formalization involved establishing three distinct groups to coordinate different parts of the overall release. I have already described the first group, a distributed group of build managers assigned to different large components of Beta, such as Betalite.

The second group focused on subsystem integration and test. This group took subsystems such as Betalite and built it with another complete subsystem and tested their interactions. This group did these one-on-one subsystems tests for all the subsystems. The purpose of this intermediary stage was to test interactions between the subsystems prior to the third stage of release management, which was the process of bringing together the whole system for testing and subsequent shipping to customers. Beta's release group did this.

Creating these groups for Beta was inevitable. Prior to having a full-time staff devoted to the coordination of Beta the overall assembly of the product was almost impossible. The project was growing at such a rate that informally it was hard for anyone to know who was working on it and how all the pieces would fit together. The creation of Beta's integration and release groups solved this problem for the development staff who now knew that there were people responsible for building their code.

The creation of official release groups for Beta was more problematic for the people assigned to be in them. The first problem for those working on the release was figuring out what components made up Beta and how they were related. Beta, like many large software products, cannot be described with the use of a single architectural diagram. Each architectural diagram captures a facet of the structure of the product such as the subsystem structure, the run-time architecture, the hardware components, and so forth. From these diagrams and visits to the development leaders they slowly managed to piece together the information they would need to build the test suites and order the sequence by which the product was built.

The second problem stemmed from the formalization of this process itself. The three groups worked in a chain: the build managers passed code to the subsystems integration group, who then passed their code onto the release group. By the time the code got to the release group it was very far away from the development staff. This, in turn, led to real difficulties fixing problems in that final stage.

Sometimes these problems were caused by complex three-way interactions among subsystems. These had been missed in the second stage of testing because of the focus on one-on-one subsystem testing. However the problems occurred, when the release group discovered difficulties they were faced with the problem of determining who to report the problem too. They did not know the developers responsible and probably were not geographically collocated with them. As a result they immediately had to report the problem to a fairly senior level of management who would delegate the challenge of finding who and what caused the problem down into their own organizations.

Consequently, problems that wound up in the final stages of the release process became highly visible within the organization. This put a lot more pressure on the developers to get the problem fixed and if possible ensures that their code was not responsible. It created a difficult environment to debug dependencies among subsystems since the nature of cooperation was changed from being a cordial team effort to being an environment where the problem was highly visible.

Moreover, it was not a problem that developers could easily resolve by unit testing their code more rigorously prior to entering it into the build. Since the bugs found at the final stages often involved spurious interactions among subsystems that did not always appear to be related, it was not a dependency that developers even thought they had. When they did know that they shared such a dependency it was difficult for them to coordinate the changes with their distant partners. For example, sometimes they would not have the name of the corresponding developer, sometimes that person would be several time zones away, and more importantly they rarely had easy access to the current version of the code from the other group to test their own work against.

5.2. SOFTWARE ARCHITECTURE: UNOFFICIAL COORDINATION ROLE

Individuals working in build and release groups have an organizationally established role that focuses on coordinating change among large software systems. The architects at Computer and Comms Corp did not have that role officially, but it constituted a considerable percentage of their day-to-day work. In fact, in both companies it was impossible for them to become a "good" architect without coordinating work among the groups responsible for developing their designs.

At both Computer and Comms Corp architecture was a distinct function within the development life cycle. At both companies, architects designed new features and enhance their existing range of products as well as design new ones for emerging markets. They work hard to define the overall product structure, and then get involved in breaking down the sections into subsystems that different groups can work on. Because they usually start from existing code, the architects often find themselves in the position of mapping out the new extensions and directions for the products, in technical terms, and then assigning the work to the appropriate groups. As this architect put it,

> the job isn't so much thinking up new architectures but getting them accomplished. . . . that's the larger part of the effort . . . in many ways the act of coming up with a specific architecture is you know just draw some lines and boxes and arrows and it's almost a dime a dozen. Lots of people have intellectual architectures and not too many people can translate those into actions and agreement and creation, that's really where the rubber meets the road. Architect, Computer Corp.

The role of the architects at both companies was to turn a problem, often ill defined at best, into a solution and find development groups to build their solution. The first step for the architects at Comms Corp was to assemble a team of people to help work on aspects of the solution space.

The explanation of why the architects assembled a team to work on the solution was always given as deriving from the fact that a single architect could not produce the best overall design. Architects and their managers felt that each architect was a specialist in aspects of design and in certain products so by working in a team they would broaden their skill set and produce better designs. The architects chose team members with that in mind.

The teams that the architects assembled consisted of two kinds of individuals: core team members and consultants. Core team members usually came from the architect's home department and attended all meetings. These individuals broadened the design space by being familiar with other products. This bought an understanding of possible interactions between the new or enhanced product being designed and other products.

Consultants did not come from the architect's department but anywhere in the company. Architects used networks of contacts (derived from their many years of service within the company) to find these consultants.[4] Consultants attended specific meetings where they contributed detailed design comments. In fact, it was the consultants who helped Comms Corp's architects transition their solutions from paper designs to tangible products. The following field note excerpt illustrates how consultants aided in that transition:

> 7:00 a.m. Architecture Meeting. The lead architect hands out slides of the current design for the controller. Each person in the room has copies of the slides. Unlike the previous meetings I've attended, the core team is expanded with three consultants, two "software specialists" (senior developers), and a hardware person.
>
> The lead architect begins presenting the solution. He begins with the highest-level overview of the product. After about 10 minutes, the hardware person stops the architect. He offers a slightly different design of the overall footprint of the product. He then relays some news that he's heard about the factory that will likely manufacture the new design. Apparently they've been swamped with orders for a similar product in terms of size and geared their production lines for that product. He suggests that one way to get this into production faster is to change the layout so that it has the same physical footprint as this other

product thus saving the factory having to change their production lines with the associated loss in time. He stresses that he doesn't think that the factory will incur the downtime to change lines because of their production goals, and urges a redesign.

The architect takes close notes and asks for a name of someone that he can call there to get the latest news. The architect continues on, moving to the software architecture. The two software specialists have been invited because it is their departments who would be the best candidates to implement the solutions. They seem agreeable to this. The architect explains how he thinks the work could be divided among the departments and the kinds of dependencies that exist between the two departments. In exchange the software specialists discuss the technical merits of the proposal with each other as well as the architect. During this discussion each developer also describes the current development plans that the department has, and how this might fit into those. Again the architect takes notes.

Architects invite consultants to meetings for much more than technical feedback on the design. In addition to gathering critical information about difficulties in the technical design, architects also gather vital information about the technical trajectories of development organizations that might implement their design. This was vital information because the software and hardware groups who might implement the design had their own schedules and commitments for months and even years in advance. One way to transition the design into development was to design a solution that fitted into these trajectories. Consultants provided that kind of design information.

This information was cycled into the design in various technical ways. For example, the news relayed by the hardware person was followed up by a call to the factory to learn more about the current production schedules. The architects then followed up with a meeting of the core team members where the lead confirmed the news from the factory and they spent the next hour reworking the design to take the suggested form. They did this to increase the chances that the factory would be able to work the new product more easily into their production plans.

The two software specialists invited to this meeting had attended previous meetings individually. In previous meetings the architects had worked with each specialist on designing the software that the specialist's department would build. The architects called this process "buy-in." Buy-in from all departments involved meant involving the developers early and often by inviting influential developers to their meetings to work on the design. Influential developers were those people who could take the design back to their departments and through their communication skills and standing within their own organization could convince others that this was the right solution (Curtis et al., 1988). By bringing the influential developers into the design process the architects encouraged those developers to feel like they had contributed to the solution and be invested in seeing it get implemented.

Architects at Comms Corp told me that without this kind of buy-in products would not get built; instead, during the transition from design to development they would fall apart. No one within the candidate development groups would make the design a high-priority and it would be left to compete with either a solution designed by the development department itself or other work commitments.

In this meeting, the software specialists, now committed to their individual parts were bought together. The architects did this to see whether the software groups could work together. Without the necessary collaboration among development departments the architects' solution would unlikely be implemented. At both Comms and Computer Corp, architects could tell me the names of departments that could not work together easily or at all.

> That kind of thing can happen and when you do have a pair of groups who have this dependency it becomes difficult to accomplish change and when change is really needed in order to deal with technology advances with competitive requirements things like that we need to kind of revisit the interfaces between the software components that those two groups produce and as a result we need to revisit the interfaces on a personal level. Architect, Computer Corp.

If the solution consisted of an enhancement to an existing product or required the coordinated efforts of multiple independent development organizations then the architects had to factor this into their matching process.

Meetings like the one above and the many others held throughout the course of a design show how architects who were formally working on the process of decomposition were also working informally on the process of recomposition. While designing a solution space that could be separated and implemented by departments, they were also facilitating relationships that spanned hardware and software, trying to make all the connections visible to all the people involved. The architects did this by introducing different constituents in the overall product design to each other.

6. Recomposition in the face of change

In the previous two sections I have focused on how individuals in organization use communications and their role within the organization to either formally or informally engage in recomposition work. All the time individuals engage in recomposition, the circumstances of the organization around them change. For example, when Beta grew in size it acquired a formal release process that made some dependencies visible to senior management in new ways. In this section, I describe a dramatic organizational change with consequences for one project I call Shape.

Shape was a particularly interesting project because it was a reuse project. Software reuse is the process of building a piece of software once and then reusing that product inside others. The advantages claimed are that by reusing software a

development effort receives reliable software quickly and cheaply. The software is more reliable because it has been built and rigorously tested once and now can be simply adopted by another group and incorporated into their product. Of course, reusing software means that many projects come to technically depend on what they reuse. This was certainly true for Shape and its customers.

Shape was launched with several other reuse projects about seven years ago. The corporation set up a division of the company that contained all the reuse projects and funded it by levying a "tax" on the product development groups. However, after a couple of years the corporation changed its priorities away from reuse and reorganized. The reuse division was dismantled and all the reuse projects were encouraged to follow their largest potential customers into the same development division. At the same time as the reorganization occurred the financial model for reuse changed. Instead of all development groups being "taxed," reuse projects were told that they would need to raise revenue directly by charging internal customers for reusing Shape.

Unsurprisingly, the architects were among the first people who changed their use of Shape. Project Delta was one such architecture project. Delta incorporated a technology called Middle that in turn reused Shape. Field notes from the first meeting after the corporate announcement was made illustrate how restructuring can influence relationships:

> 10:30 a.m. Project Delta Architecture Meeting. This is not a scheduled meeting. The lead architect arranged the meeting as a result of the recently announced reorganization. All the core team members are present along with two representatives one from Middle and one from Shape. The lead architect hands out slides with the current design on it. Along side the designs is a print out of the e-mail that outlines the reorganization. At this time some of the implementation details about the reorganization are not written out but all of the people in the meeting have been through at least one such reorganization prior to this.
>
> The architects and representatives review the materials. After a short period of reviewing both sets of materials the discussion begins. First, they discuss the likely implications of the restructuring. In particular the Shape representative tells everyone that she believes that the project will be moved into Division A. She bases this on what her manager has told her and the fact that Shape's largest customer (who will pay the most) has moved into Division A too. The representative from Middle then says that he believes that their project is going to stay in Division B. The architects working on Delta believe that their project will be funded and built in Division B.
>
> The meeting then begins to examine the reasons behind the creation of Divisions A and B. The discussion focuses on the technical trajectories of both divisions. What products are in each division, what can be deduced about whether the technical directions of A and B will diverge. Divergent technical directions will make Shape less and less reusable inside products headed in a different technical direction.

> After gathering this information, and lots more speculation besides, they turn to the question of whether the funding can be raised in Division B to pay Division A to use Shape. The answer comes in the form of examining the new organizational chart. It appears on the chart that the most senior levels of management have to be involved to gain a funding commitment. While that's not impossible, it seems improbable. The end result of the meeting is that after six months of work they abandon the design that uses Shape. The architects thank the Shape representative for her help and she leaves. The lead architect decides to call a meeting of the core team to start the process of reworking the architecture.

Within a month of this meeting Delta had completely changed. Delta abandoned Shape because of the concerns about future technical direction and potential financing. The architects then searched for another solution to avoid completely reworking the solution. They found it in the form of a recent acquisition that happened to be in Division B. Unfortunately, the acquired company's product also removed the need for Middle too. The architecture of Delta was reworked and implemented on top of the acquired company's product and Middle was quietly abandoned.

It was this meeting that first alerted me to the significance of corporate change on recomposition. In this case, a change in the corporation had introduced technical and financial divisions that were considered too difficult to work with. It was challenging enough that the architects and representatives thought the best solution was to abandon six months of design work because the coordination required to ensure that all the pieces could be assembled was too difficult.

It was this meeting that introduced me to Shape. I decided I wanted to follow up with Shape to see how their customer base had changed since the restructuring. The opportunity to do so came through an invitation to participate in a Shape project retrospective where 15 people from the project held a daylong meeting to discuss the lessons learned from the previous year on Shape.

Unsurprisingly, the restructuring took up a considerable part of the discussion. Within a few weeks of the announcement Shape's customer base had changed dramatically. First, there were the projects like Delta, and others that were past design and into development, whose management decided to stop using Shape. For many it was not an overnight decision in the sense that according to Shape developers they were going to transition from Shape to their own development efforts. In other words, these projects took one last "free Shape" and now had development plans that included building their own Shape-like functionality.

Another change to Shape was the arrival of new customers. When it was known that Shape would join Division A, a variety of small projects in Division A decided that they could afford to take advantage of Shape in their own code. Although Shape had been free before, some development projects had felt that Shape was too distant from their actual needs and it would be more effective to build their own Shape function. The arrival of Shape in Division A, and the implication that

that would focus Shape on to concerns that permeated all of the projects in Division A, attracted smaller Division A projects.

The project retrospective concluded with an analysis of the changing trajectory of Shape itself. Shape had finally abandoned the premise that its functionality could be widely reusable. Instead its functionality now focused on technical objectives of all the products that comprised Division A. Those developers recognized that it was a fear of this close technical alignment with the largest Division A customer that Shape originally followed into Division A that had driven many other potential reuse customers outside the Division away. One the other hand, they recognized that they had also gathered new customers interested in reusing Shape, precisely because it was now a more focused effort.

7. Discussion

Recomposition is all the coordination required to manage the dependencies among software components. In the previous sections I outlined some techniques individuals use to manage this coordination: communication through email and meetings, and informal and formal organizational roles. I also described how environmental circumstances, such as corporate change, influence recomposition work.

The need for recomposition arises from the association between the process of decomposition and the division of labor. Decomposition does not just separate the software system into components, but also creates work for the individuals assigned those responsibilities. However, while modular decomposition has received much attention among software engineering researchers, the coordination of the process across the division of labor has received far less within the literature. When it arises, it is often in the form of discussion about how various team and organizational structures should reduce the need for communication among developers (see for example, Brooks Jr., 1995; Sommerville, 1996). In other words, the coordination overhead that the division of labor creates is something that must be designed to be minimized (which is what Parnas (1972) proposed with modular decomposition).

So, while the theory of software engineering proceeds with the goal of reducing communication among and across teams and organizations, the practice of software development remains highly communicative. The examination of recomposition as practiced in real software development projects reveals a number features of the work that may shed light on the difficulties of trying to reduce communication as well as offering other alternatives for managing dependencies. Specifically, in the remainder of this section I describe three features of software development work that are revealed by recomposition: the affects of environmental disturbances on development work, the types of dependencies that require recomposition, and the images of organizations required to manage the recomposition.

8. Environmental disturbances

First, recomposition work makes visible the articulation of problems that might be described as environmental disturbances that disrupt development. Environmental disturbances are disruptions to the production process that come from outside that process. For example, the Network Project meeting revealed two sources of environmental disturbance that delayed development: waiting for external vendors or internal development departments to deliver hardware and software. Delta and Shape experienced another example of an environmental disturbance, one generated by the corporation in the form of a reorganization that changed who they could work with easily by altering financial arrangements and technical trajectories.

The presence of these kinds of environmental disturbance generates the need for some recomposition work among those affected by the delays and changes. These disturbances also come with a cost to the production process because they create instability in that production process. In particular, organizations that depend on a stable sequence of production processes often work hard to protect that work from environmental disturbances (Thompson, 1967). An assembly line in a manufacturing plant is an example of a production process that requires protection from environmental disturbances. Organizations can and do use "buffering strategies" such as having groups to manage the procurement of inputs (purchasing), and the distribution of outputs (sales), and many other options (Scott, 1992).

The production of software is typically described in terms of stable sequences of processes. These sequences of processes are often known as the software development life cycle (see for example, Boehm, 1988; Royce, 1970; Sommerville, 1996). Moreover, software development corporations use buffering strategies to protect themselves from environmental disturbances, such as purchasing and sales. However, software development seems to have other sources of environmental disturbance that still affect the production process.

Software engineering researchers have long known of one type of environmental disturbance: requirements volatility (Curtis et al., 1988; Sommerville, 1996). Requirements volatility occurs when, for example, market forces or clients' requirements change during the development process forcing modifications to the software. Software engineers have tried to accommodate this type of environmental disturbance into their life cycle models by recommending that requirements are revisited multiple times and/or eventually freezing the ability to change the requirements. The presence of recomposition work suggests that requirements volatility is not the only source of environmental disturbance that impedes development. Disturbances in the dependencies on external and internal sources for inputs into the development process create delays and difficulties.

Moreover, it highlights the fact that organizations are not always the passive recipients of dependency disturbances; indeed, they can generate them themselves. Specifically, reorganizations can change who depends on whom for what and how easy that will be to coordinate. A focus on recomposition provides some rationale

for why considerable lateral, informal, communication plays a critical role in managing environmental disturbances, legitimizing and providing the means for accounting for this type of work within software development. It may also provide a starting point for considering the application of buffering strategies not yet considered by software development organizations.

8.1. TYPES OF DEPENDENCIES

Second, recomposition work focuses on the types of dependencies that individuals are trying to coordinate. For example, Alpha meetings surfaced dependencies among the development teams who needed to be made aware of the adjustments to other pieces so that they could incorporate those changes into their own work. In addition to providing status on their own pieces, team leaders needed to know about their colleagues' components. In other words, there was a mutual dependence among the team leaders.

Sometimes coordinating dependencies via broadcast surfaced unknown mutual dependencies. Occasionally people would e-mail back the initial developer and inform them of reasons why they could not make that change. In fact it was possible for a developed to share a mutual dependency that they did not even recognize as such. The role of build and release managers often mediated these mutual dependencies for developers who shared them.

Organizational theorists interested in how organizations organize dependent tasks have made distinctions between different types of dependencies based on how much work it takes to coordinate them. In Thompson's (1967) classification "reciprocal" dependencies are those where two (or more) pieces of work mutually depend on each other. He argues that while other kinds of dependencies may be managed through rules, routines, plans and schedules, mutual dependencies require communication to coordinate.

Galbraith (1977) suggests a number of lateral coordination mechanisms by which organizations can communicate their most complex dependencies. One solution, liaisons, focuses on having a special position or group designed to facilitate interchange among dependent groups. The recomposition work of build managers suggests that they are liaisons among multiple developers working on related code. The architects also serve as liaisons among multiple developers working on related components. Although it is not a formal requirement of their job, to coordinate dependencies, it is a practical requirement of their work. Indeed, without engaging in recomposition work to coordinate all the parties that will work on their design, it is impossible to realize the paper design as a product.

Both the build managers and the architects illustrate the power of liaisons to coordinate shared dependencies. Beta's release group illustrates some of the difficulties of the same approach. Unlike the build managers and architects who could communicate and coordinate with individuals who shared a dependency, the release

groups did not always know who was responsible for the code that shared a mutual dependency. In the absence of this, their liaison role was limited by their need to make any problems with dependencies highly visible within the organization.

The software engineering literature recognizes the technical difficulties of dependencies. The focus on reducing coupling, for example, recognizes the technical value of creating independent components. This attempts to reduce dependencies from reciprocal to something similar to what Thompson characterizes as pooled interdependencies (Thompson, 1967). Pooled dependencies contribute to the whole but have no direct connection to each other. Interface specifications and information hiding allow independent groups of developers to pursue their own development work with only the knowledge of what functionality is provided and how that functionality is called. When this works, it is the common standard that they share that provides the coordination required.

A focus on recomposition suggests that many reciprocal dependencies remain, even when some have been reduced to pooled dependencies. Recomposition work makes those dependencies visible for organizational analysis. The dependencies do not appear as just technical problems, but also as mutually dependent work activities that require communication and coordination.

Practical attempts to cope with the recomposition work created by mutual dependencies have tried informal and formal organizational solutions that allow those responsible for the dependencies to coordinate with each other. These examples may suggest that recomposition is most successful when the individuals have direct contact with each other, and without the intervention of the formal organization. The focus on recomposition highlights organizational theoretic aspects of dependency management, and suggests a direction for corporations to pursue if they wish to coordinate mutual dependencies: examining mechanisms for facilitating lateral communications beyond liaisons.

8.2. ORGANIZATIONAL ANALYSIS IN PRACTICE

Third, recomposition requires organizational thinking by the individuals trying to coordinate their efforts. Previously I argued that recomposition makes the organization visible for analysis, mainly in those cases by researchers and practitioners interested in understanding and improving software development. However, recomposition work also shows that the people who manage dependencies must also engage in organizational analysis. They need to develop an understanding of their organization in order to coordinate their dependencies. In fact, the developers, architects, and managers used a number of metaphors of organization to explicitly talk about and implicitly manage their recomposition (Morgan, 1986). These metaphors helped the individuals manage recomposition work.

The first metaphor of organization in use at all three organizations was of organization as machine. This metaphor decomposes an organization into a set of interlocking elements that work together. Individuals used the machine metaphor to

think about recomposition. For example, the architects used information about the structure of the organization to make a decision about whether to continue using Shape in Delta. They used the reorganization announcement to made deductions about what kinds of technical directions projects in Divisions A and B would take. They also looked at the chain of management up from Shape and Delta. They were searching for the first point of common management between Delta and Shape to determine how easy it would be for Delta to pay Shape. When they discovered that it was at the level immediately below the CEO the architects decided that it would be very difficult for Delta to continue using Shape.

Beta's release group used the formal structure of the organization much less successfully. The distance between the release group and the development organizations was not just a matter of lines on the organization chart, but also a gap in their knowledge about who to communicate with. Left only with a formal means of communicating that someone had failed to manage a dependency this failure became known to their management who communicated upward and across to the development managers who then had to search their own organizations for resolution. This made any problem very visible, which violated expectations of how people reported software failures. Many of these expectations were grounded in the metaphor of the organization as a culture, with norms and practices.

This was most explicit in the Alpha meetings. The culture of Alpha came with specific expectations about how dependencies were discussed. Team leaders knew that difficulties should be resolved outside the meeting with the other person responsible, and that the department head should be informed. The department heads had a slightly different set of cultural norms associated with managing changes across departments. They did not ever raise them to the level of the Vice President. Instead they talked directly to the other department head. It is this "knowing" of the practices that smoothed some of the potential difficulties surrounding recomposition. Indeed, the Alpha department head also illustrates the multiple cultural practices within the Network Project. Specifically, she moved between two different settings where she used two slightly different sets of practices to manage the reporting of delays resulting from dependencies that needed to be resolved.

The people of Alpha used a third metaphor. The influence of the environment on the organization is Morgan's metaphor of organism, which comes from a biological perspective that a system can only sustain itself in terms of its environment. For Alpha, the environment was not always good at sustaining them internally. The environment was full of third party vendors who delayed shipment of hardware and software that the Alpha staff needed including chips, processors, operating systems, compilers and debuggers. Alpha staff articulated their schedule delays as dependencies on these outside inputs.

The final metaphor that the architects used was an understanding of the organization as a political system. This metaphor emphasizes how individuals and groups

use various resources to control outcomes. The architects both saw politics in other parts of the organization and made use of political means in their own work.

They saw politics in development organizations that could not work together. They used their insight into these difficulties in their design process. They tried to avoid technical designs that made the best qualified to implement departments be groups who could not work together. Although I did not see this phenomenon occurring in the design meetings that I attended the architects I interviewed spoke about this in terms of political rivalries.

The architects also used their own and their colleagues' networks of contacts to gather and apply information to their design. For example, they found consultants to their design projects from all over the company. The information that these consultants provided helped the architects better ensure the outcome of their design was an implemented product. This was essential to their own success, inside their department at merit review time, and externally as being someone who other groups wanted to help with the design of enhancements or new products. Like any good politician, the architects worked hard to maintain those networks.

Faced with the reality of needing to coordinate dependencies, software practitioners use metaphors of organizations to guide their attempts. A focus on recomposition allows us to see when and how successfully software practitioners apply different metaphors to manage their dependencies. The use of metaphors, particularly in the development of solutions to coordination difficulties offers an opportunity to see how those doing development work conceive of their own circumstances. For individuals, such as the Network Project department heads and the architects, the organization that they needed to understand and work within was substantial and complex, spanning formal structures, geography, culture, and multiple technical trajectories.

9. Conclusions

This paper described "recomposition" which is all the work necessary to coordinate the assembly of software systems. The need for recomposition arises when the software problem is decomposed into component solutions. Recomposition is the process of coordinating the assembly of these pieces so that they can become a whole software system, and it is made harder because the pieces have mutual dependencies.

The paper emphasized how individuals working on large projects engage in many forms of recomposition as part of their day-to-day work. Individuals communicate through e-mail and meetings, and others have special roles designed to facilitate recomposition. Still others, architects, have roles that involve recomposition informally, even though it is not part of their job description. I also described how changes in an organization could change the nature of recomposition.

What makes recomposition hard is the dependencies among those components, that manifest themselves in relationships among individuals that need to

be coordinated. These relationships are the layer of administration that comes as a result of the division of labor (Scott, 1992). Beyond the definition of recomposition – making all the work to coordinate dependencies visible – re-establishes the connection between the software components and the division of labor. The perspective offers a perspective on software development that makes some of the sources of coordination difficulties visible and available for analysis. Specifically, I described three features of software development work that are revealed by recomposition: the affects of environmental disturbances on development work, the types of dependencies that require recomposition, and the images of organizations required to manage the recomposition.

Acknowledgements

I would like to thank all the people who have talked to me about software development over the years. This paper is dedicated to Nancy.

Notes

1. It is worth observing here that many other communities also study software development work including human-computer interaction researchers interested in design work, empirical studies of programmers interested in questions of cognition in individual programmers, social scientists interested in the production of software and not least software engineering researchers concerned with the improvement of practice through process and tool enhancement.
2. A configuration management system is typically constrained by the code that the database can hold. Modern CM systems use database technology to manage all the code and manage the relationships among the components. Therefore the constraints tend to be how much code the database can hold and how many people it can support accessing the code base at the same time. It is the lower of these two bounds that determines how large the group using the tool can be.
3. Cusumano and Selby (1995) describe the consequences of being known as the person who broke the build at Microsoft as being the person who wore a special hat and also had the job of coordinating all the build work until the next build broke.
4. All the architects at Comms Corp had worked with the company for at least a decade. Their experience was typically as a developer on a product that they now produced architectures for. They were often well known individuals within the corporation who had been promoted technically into their architecture role.

References

Boehm, B. (1988): A Spiral Model of Software Development and Enhancement. *IEEE Computer*, vol. 21, no. 5, pp. 61–72.

Brooks Jr., F.P. (1995): *The Mythical Man-Month: Essays on Software Engineering 20th Anniversary Edition*. Reading, MA: Addison-Wesley Publishing Company Inc.

Bucciarelli, L.L. (1994): *Designing Engineers*. Cambridge, MA: MIT Press.

Button, G. and W. Sharrock (1996): Project Work: The Organisation of Collaborative Design and Development in Software Engineering. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 5, no. 4, pp. 369–386.

Conway, M.E. (1968): How Do Committees Invent? *Datamation*, vol. 14, no. 4, pp. 28–31.

Curtis, B., H. Krasner and N. Iscoe (1988): A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1287.

Cusumano, M.A. and R.W. Selby (1995): *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York, N.Y.: The Free Press.

Galbraith, J. (1977): *Organization Design*. Reading, MA: Addison-Wesley.

Ghezzi, C., M. Jazayeri and D. Mandrioli (1991): *Fundamentals of Software Engineering*. Englewood Cliffs, N.J.: Prentice-Hall.

Grinter, R.E. (1998): Recomposition: Putting It All Back Together Again. In D.G. Durand (ed.): *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '98)*, Seattle, Washington, November 14–18. New York, N.Y.: ACM Press, pp. 393–403.

Herbsleb, J.D. and R.E. Grinter (1999): Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, Los Angeles, CA, May 16–22. ACM Press, pp. 85–95.

Herbsleb, J.D., H. Klein, G.M. Olson, H. Brunner, J.D. Olson and J. Harding (1995): Object-Oriented Analysis and Design in Software Project Teams, *Human-Computer Interaction*, vol. 10, nos. 2–3, pp. 249–292.

Herbsleb, J.D., A. Mockus, T.A. Finholt and R.E. Grinter (2000): Distance, Dependencies and Delay in a Global Collaboration. In D.G. Durand (ed.): *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*, Philadelphia, PA, December 2–6. New York, N.Y.: ACM Press, pp. 319–328.

Mockus, A., R.T. Fielding and J.D. Herbsleb (2000): A Case Study of Open Source Development: The Apache Server. In H. Gall (ed.): *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 4–11, June. IEEE Press, pp. 263–272.

Morgan, G. (1986): *Images of Organization*. Sage Publications, Inc.

NATO (1969): *Working Conference on Software Engineering*, Report, NATO Scientific Affairs Division.

Palen, L. (1999): Social, Individual and Technological Issues for Groupware Calendar Systems. In *Proceedings of the ACM Conference on Human Factors in Computing Systems CHI '99*, Pittsburgh, PA, May 11–13. ACM Press, pp. 17–24.

Parnas, D.L. (1972): On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058.

Parnas, D.L. and P.C. Clements (1986): A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251–257.

Perry, D.E., N.A. Staudenmayer and L.G. Votta (1994): People, Organizations, and Process Improvement. *IEEE Software*, vol. 11, no. 4, pp. 36–45.

Potts, C. and L. Catledge (1996): Collaborative Conceptual Design: A Large Software Project Case Study. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 5, no. 4, pp. 415–445.

Royce, W.W. (1970): Managing the Development of Large Software Systems. In *Proceedings of the IEEE WESCON*, Los Angeles, CA. IEEE Press, pp. 1–9.

Scott, W.R. (1992): *Organizations: Rational, Natural and Open Systems*, 3rd edn. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Sommerville, I. (1996): *Software Engineering*, 5th edn. Menlo Park, CA: Addison-Wesley Publishing Company.

Strauss, A. (1987): *Qualitative Analysis for Social Scientists*. New York: N.Y.: Cambridge University Press.

Strauss, A. and J. Corbin (1990): *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Newbury Park, CA: Sage Publications, Inc.

Tellioglu, H. and I. Wagner (1997): Negotiating Boundaries: Configuration Management in Software Development Teams. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 6, no. 4, pp. 251–274.

Thompson, J.D. (1967): *Organizations in Action*. New York, NY: McGraw-Hill.

Whittaker, S. and H. Schwarz (1999): Meetings of the Board: The Impact of Scheduling Medium on Long Term Group Coordination in Software Development. *Computer Supported Cooperative Work (CSCW): The Journal of Collaborative Computing*, vol. 8, no. 3, pp. 175–205.

Yates, J. (1989): *Control through Communication: The Rise of System in American Management*. Baltimore, Maryland: The Johns Hopkins University Press.